# 操作系统原理实验一

## 一、安装配置

1. 电脑系统是macOS，所以网站上老师配好对Ubuntu虚拟机不能用，一开始就也用的VMware创建虚拟机，共享原系统文件lab1，但后来发现室友用的Parellels真方便，就投向了Parallels的怀抱。Parallels里共享文件要比VM方便很多，也不会有下面的要求，当时下面的注意我发现的很晚，发现后又把文件拿出来重新配置，很累人。

### 构建并安装

> **注意！**
> 该过程 **不能** 在 Windows 目录下完成！ 如果使用 WSL，务必注意不要将其放在 `/mnt/` 目录下；如果使用 VMware，务必注意不要将其放在 `/mnt/hgfs` 目录下。

2. Make qemu完后，发现《常见问题汇总》里说的SDL support的问题，检查发现自己也是no，然后按操作make clean，再重新make，然而一直有Werror错误，但makefile重装了就不会有Werror，连环扣一直报错，当时时间有点晚脑子木掉了，直接强制clean把虚拟机删掉重装了。不过这个错误直接把lab1文件重制，然后再检查local里的qemu文件是否存在就能解决，算是clean不干净的手动清除。
3. 我没发现qemu=路径那句话被注释掉了，浪费了半天时间检查系统……如下图的问答。

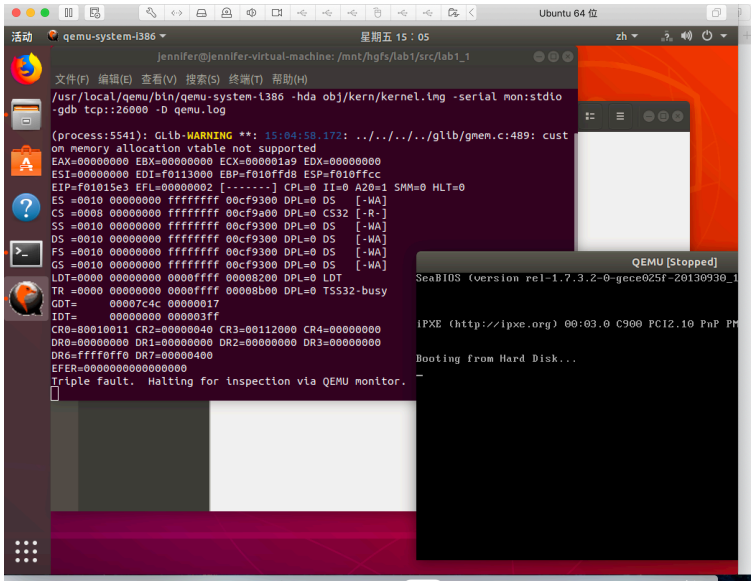
share improve this question

asked Jun 8 at 15:35
Muhammad Sufyan Raza
35 ● 6

1 The comment in the env.mk file answers your question: "If the makefile cannot find your QEMU binary, uncomment the following line and set it to the full path to QEMU". On Ubuntu the i386 emulation binary is 'qemu-system-i386'. – Peter Maydell Jun 10 at 10:09

Yes it worked perfectly!! Thanks – Muhammad Sufyan Raza Jun 10 at 13:50

add a comment



4. Triple fault那里卡了三天，然后发现qemu版本问题，重新clone了新的qemu，然后询问同学，在GUNmakefile里加了CFLAGS += -fno-pic，通过。（当时没有发现网站上还有另一版本的问题汇总，自己以为两个问题汇总是一样的内容所以忽略了，浪费的时间太多想哭；由于这个解决方案是传了好几个人才问到的，所以没法问最开始的大佬这样做的原因，在网上查了些资料，只知道-fno-pic是显试指示）

```
PERL    := perl

# Compiler flags
# -fno-builtin is required to avoid refs to undefined functions in the kernel.
# Only optimize to -O1 to discourage inlining, which complicates backtraces.
CFLAGS := $(CFLAGS) $(DEFS) $(LABDEFS) -O1 -fno-builtin -I$(TOP) -MD
CFLAGS += -fno-omit-frame-pointer
CFLAGS += -Wall -Wno-format -Wno-unused -Werror -gstabs -m32
CFLAGS += -fno-pic
# -fno-tree-ch prevented gcc from sometimes reordering read_ebp() before
# mon_backtrace()'s function prologue on gcc version: (Debian 4.7.2-5) 4.7.2
CFLAGS += -fno-tree-ch

# Add -fno-stack-protector if the option exists.
CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-
protector)

# Common linker flags
```

---

## 二、实验操作及问题解答

下面是练习解答和吐槽的杂糅，因为吐槽是由做练习的过程中产生的，我也不太清楚该怎么直接吐槽，所以连带着没有吐槽的问题一起说了。

lab1其实没有很难的点，但是我在这次作业上耗费的时间非常长，从老师布置完作业我就开始着手做，直到国庆假期才做完，经常是走一步卡一步，当时在环境配置完的时候还松了一口气，后来发现自己很天真。

手里的四版实验指导差异太大，练习题有的多有的少，中文版有的题的翻译让人摸不着头脑，我只能在各种实验指导里来回横跳。实验指导有些部分意义不明，有很多次卡了半天翻了翻另一版说明就恍然大悟，练习和作业写代码的位置也要自己翻来找去。

### (一) 2.2 Boot Loader

这里主要是熟悉GDB指令，顺着问题查看文件特性，就是问题大多数说的都不具体，得结合多版说明文档和网络教程才能搞清楚它到底想让我做哪些事，想要什么结果。

#### 问题1 解答：

1. 处理器从何时开始执行32位代码？究竟是什么导致了从16位模式到32位模式的转换？

在运行到 *0x7c00断点处，然后si逐行查看，发现在[ 0: 7c2d ]时转为32位模式。

```
(gdb)
[   0:7c26] => 0x7c26:  or     $0x1,%eax
0x00007c26 in ?? ()
(gdb)
[   0:7c2a] => 0x7c2a:  mov    %eax,%cr0
0x00007c2a in ?? ()
(gdb)
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7c32
0x00007c2d in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c32:      mov    $0x10,%ax
0x00007c32 in ?? ()
```

2. 引导加载程序执行的最后一条指令是什么?它刚刚加载完的内核的第一条指令是什么?

引导加载程序的最后一条指令是boot/main.c中bootmain函数最后的((void (*)(void)) (ELFHDR->e_entry))(); 这个第一条指令位于/kern/entry.S文件中，第一句 movw $0x1234, 0x472

```
// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry))();
```

```
entry:
        movw    $0x1234,0x472                   # warm boot
```

3. 内核的第一条指令在哪里?

上面2题写了，在/kern/entry.S里

```
parallels@parallels-Parallels-Virtual-Platform: ~/lab1
File  Edit  View  Search  Terminal  Help
parallels@parallels-Parallels-Virtual-Platform:~/lab1$ objdump -x obj/kern/kerne
l

obj/kern/kernel:     file format elf32-i386
obj/kern/kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Program Header:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
         filesz 0x000074be memsz 0x000074be flags r-x
    LOAD off    0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
         filesz 0x0000a300 memsz 0x0000a944 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
```

4. 引导加载程序如何决定必须读取多少扇区才能从磁盘获取整个内核?它在哪里找到这些信息?

执行了几条objdump指令，多少扇区的信息我没找出来……

## (二) 2.2.1——2.2.3

2.2.1主要工作为复习指针知识，同时了解系统从实模式到保护模式的意义，了解-h和-f指令。2.2.2中修改地址make clean查看boot无法启动：

```
Program received signal SIGTRAP, Trace/breakpoint trap.
[   0:7c30] => 0x7c30:  ljmp   $0x8,$0x7c36
0x00007c30 in ?? ()
```

在boot.s中找到对应代码：

```
# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp    $PROT_MODE_CSEG, $protcseg
```

### 2.2.3问题解答：

1. 解释printf.c和console.c之间的接口，具体来说，console.c导出什么功能？printf.c如何使用这个函数?

打开console.c文件可以观察到最后的cputchar(int c)，和printf.c共用，这也就是它们的接口。在cputchar中调用cons_puts函数，在向上查询可以看出console.c导出了IO端口的操作，定义了字符的显示功能。printf.c可调用cputchar函数。

2. 解释下面console.c中的代码片段

```
1       if (crt_pos >= CRT_SIZE) {
2               int i;
3               memcpy(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE -
CRT_COLS) * sizeof(uint16_t));
4               for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5                       crt_buf[i] = 0x0700 | ' ';
6               crt_pos -= CRT_COLS;
7       }
```

memcpy和文件里的memmove函数都是用来拷贝字节的，知道函数的意义也就可以知道这段代码的意义，pos超过size则少一部分拷贝字节，当前位置回档，也就是说：如果写到最后一行没有空地了，则全部内容向上滚动一行，留出一行空行。

3. 在调用cprintf()时，fmt指向什么?ap指向什么?列出(按执行顺序)。对cons_putc、va_arg和vcprintf的每个调用。对于cons_putc，也列出它的参数。对于va_arg，列出调用前后ap指向的内容。对于vcprintf，列出它的两个参数的值。

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

```
x 1, y 3, z 4
```

我是谁我在哪我要把这段代码放进哪个文件哪个函数？我一开始都把这些代码在Clion上跑的……后来才知道要放在monitor里跑。。。但是，这个代码我懂，问题不太会答。

看到一个解答：fmt指向的是现实信息的格式字符串“x %d, y%x, z %d\n”，ap是va_list类型的，这个类型专门用来处理输入参数的个数是可辨的情况，所以ap指向所有输入参数的集合。cprint调用vcprintf函数，并将fmt和ap传给了它，vcprintf又调用vprintfmt，来解读fmt内容。这里对va_arg进行了一次调用，调用前ap中包括x，y，z三个参数的内容：1，3，4。调用完成后只剩下y，z的内容：3,4。

4. 输出是什么?按照前面的练习一步一步地解释这个输出是如何得到的。这是一个ASCII表，它将字节映射到字符。输出取决于x86是little-endian这一事实。如果x86是big-endian，为了得到相同的输出，您会将i设置为什么?您是否需要将57616更改为不同的值?

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

```
He110 WorldK>
```

输出是He110 World，因为没有换行所以’K’是后面的。首先“H%x Wo%s”中“%x”是无符号十六进制数，57616的十六进制是“e110”；“%s”是格式化字符串，0x00646c72的ASCII码对应是rld，所以得到“He110 World”。

如果是大端（尾端放高地址处）i=0x726c6400,57616不变。

5. 在下面的代码中，’y='后面会打印什么?(注意:答案不是一个特定的值。)为什么会这样?

我用Clion跑了一下是0，qemu如下：

```
x=3 y=-267380292K>
```

## (三) 2.3.2 控制台的格式化输出

找呀找呀终于找到作业修改段。在lib/printfmt.c中case 'o'部分。作业一的帮助部分看的云里雾里，感觉参照其他函数改会有条理点。

一开始看代码根本没看懂要写的函数意义是什么，然后发现注释"//octal"，因此参照了10进制代码case 'u'改写成8进制，就只把数改了，没想到试一次就成功了，明明getuint函数和base指代的是什么还没看。

```
case 'o':
        // Replace this with your code.*****
        num = getuint(&ap, lflag);
        base = 8;
        goto number;
```

## (四) 2.3.3 堆栈

实验指导上写：

eip：当前执行指令的下一条指令在内存中的移地址；

esp：存储指向栈顶的指针；

ebp：存储指向当前函数需要使用的参数的指针。

有点迷糊，先码着，之后的变量命名也是参照的这个指针说明。

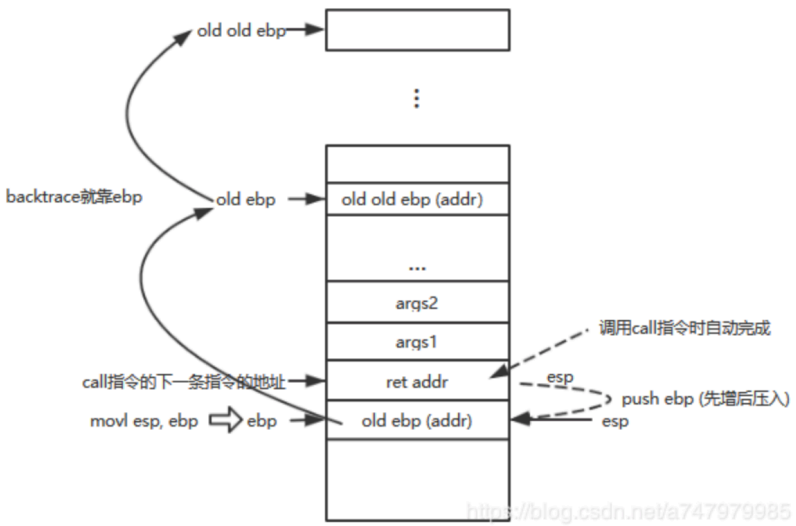
练习3:

查看test_backtrace的C代码(kern/init.c中)，完成其中mon_backtrace()，mon_backtrace的原型已经在kern/Monitor.c中。

init.c

```
void
test_backtrace(int x)
{
        cprintf("entering test_backtrace %d\n", x);
        if (x > 0)
                test_backtrace(x-1);
        else
                mon_backtrace(0, 0, 0);
        cprintf("leaving test_backtrace %d\n", x);
}
```

首先解读下test_backtrace，函数运用递归调用，在x非0时x- -重复调用自身，当x=0时调用mon_backtrace。




```
expected:
  00000000
  00000000
  00000001
  00000002
  00000003
  00000004
  00000005
```

Waring:

read_ebp(较为底层的函数，返回值为当前的 ebp 寄存器的值)

display format

```
Stack backtrace:
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2 00000031
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28 00000061
  ...
```

中间改了不少次，逻辑还是很好懂的，上手写就容易出问题，尤其是这个实验它是按输出判分的，只能按照上面的format输出，开始还怕空格不对，后来发现空格多不算错，空格少才不给分。args还是很好写的，int逐个+4就好。为了将每行都输出，还需要让ebp重新得到外围函数的值。

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
        // Your code here.********** !!!!

        int ebp, esp, eip;
        ebp = read_ebp();
        while(ebp != 0){
            eip = *((int*)(ebp + 4));
            esp = ebp + 4;

            //struct Eipdebuginfo info;
            //debuginfo_eip(eip, &info);

            cprintf("ebp %08x eip %08x args",ebp,eip);// ebp & eip
            for(int i=0 ; i<5 ; i++){ //args * 5
                esp += 4;
                cprintf(" %08x", *(int*)esp);
            }
            ebp = *((int*)ebp);// read outer layer ebp
            /*cprintf("\t%s:%d: %.*s+%d",info.eip_file
                                       ,info.eip_line
                                       ,info.eip_fn_namelen
                                       ,info.eip_fn_name
                                       ,eip-info.eip_fn_addr);*/
            cprintf("\n");
        }
        return 0;
}
```

后来发现有expected的提示写出输出感觉还是比较友好的，有次开始在测试输出的时候直接用的“%x”，也就是十进制，看到expected就改成了“%08x”十六进制输出。

## Exercise12

修改mon_backtrace函数以显示每个eip对应的函数名、源文件名和行号。

说实话，这次的修改里面dubuginfo_eip我自己真不会，看网络教程才会的。（要是看中文指导真的很迷糊，还是英文指导好，tips真的很有用）

在kern/kdebug.c里的debuginfo_eip()函数最后加上下面一段代码，加在最后是最保险的做法。

```
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
if (lline > rline)
    return -1;

info->eip_line = stabs[lline].n_desc;
```

in STABS tables. `printf("%.*s", length, string)`

```
struct Eipdebuginfo info;
debuginfo_eip(eip, &info);

cprintf("ebp %08x eip %08x args",ebp,eip);// ebp & eip
for(int i=0 ; i<5 ; i++){ //args * 5
    esp += 4;
    cprintf(" %08x", *(unsigned int*)esp);
}
ebp = *((unsigned int*)ebp);// read outer layer ebp
cprintf("\t%s:%d: %.*s+%d",info.eip_file
                           ,info.eip_line
                           ,info.eip_fn_namelen
                           ,info.eip_fn_name
                           ,eip-info.eip_fn_addr);

cprintf("\n");
```